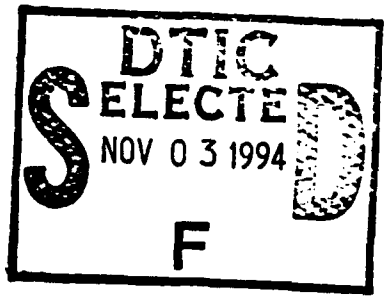
University of Southern California

Generating Examples For Use in Tutorial Explanations: The Use of a Subsumption Based Classifier

Vibhu O. Mittal and Cecile L. Paris
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695



INFORMATION SCIENCES INSTITUTE

310/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Generating Examples For Use in Tutorial Explanations: The Use of a Subsumption Based Classifier

Vibhu O. Mittal and Cecile L. Paris
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

June 1994
ISI/RR-94-384

In proceedings of ECAI 94, 11th European Conference on Artificial Intelligence, August 1994, in Amsterdam The Netherlands

REPORT DOCUMENTATION PAGE

FORM APPROVED
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	Research Report

4. TITLE AND SUBTITLE

Generating Examples For Use in Tutorial Explanations: The Use of a Subsumption Based Classifier

5. FUNDING NUMBERS

NCC 2-520 and DABT63-91-C-0025 and NSF IRI-9003087

6. AUTHOR(S)
Vibhu O. Mittal and Cecile Paris
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292-6695

8. PERFORMING ORGANIZATON REPORT NUMBER

RR-384

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

NASA AMES
Moffett Field, CA 94035

ARPA
3701 Fairfax Drive
Arlington, VA 22203

NSF
1800 G St. NW
Washington, DC
20550

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

In proceedings of ECAI 94, 11th European Conference on Artificial Intelligence, August 1994, in Amsterdam The Netherlands

12A. DISTRIBUTION/AVAILABILITY STATEMENT

UNCLASSIFIED/UNLIMITED

12B. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Examples form an integral and very important part of many descriptions, especially in contexts such as tutoring and documentation generation. Previous computational work on example generation has focused on the issue of generating valid examples in different domains. However, there are a large number of examples that can be generated for a given concept, and it is important that examples intended for tutoring situations illustrate specific features that need to be communicated. There is also a strong interaction between the examples and the accompanying textual description. The number of examples to be presented, the order in which they are to be presented, and the position of the examples with respect to the text are all dependent upon the examples generated for use in the presentation. It is therefore important that the example generator be able to generate a set of suitable examples in response to a tutoring goal in a given situation. In this paper, we present one framework, based on the use of a subsumption classifier, that facilitates the generation of appropriate tutorial examples. We illustrate the working of this framework by describing the generation of examples to illustrate the syntax in the programming language LISP.

14. SUBJECT TERMS

Tutoring systems, Text Generation, Natural Language Front-Ends

15. NUMBER OF PAGES

16

16. PRICE CODE

17. SECURITY CLASSIFICTION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reoprts. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (blank).

Block 2. Report Date. Full publication date including day, month,a nd year, if available (e.g. 1 jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classificatic in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the repor.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities.
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- Leave blank.
DOE	- Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	- Leave blank.
NTIS	- Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contins classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Generating Examples For Use in Tutorial Explanations: The Use of a Subsumption Based Classifier

Vibhu O. Mittal
Intelligent Systems Laboratory
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
mittal@cs.pitt.edu
+1-412-624-9185

Cécile L. Paris
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
paris@isi.edu
+1-310-822-1511

Abstract

Examples form an integral and very important part of many descriptions, especially in contexts such as tutoring and documentation generation. Previous computational work on example generation has focused on the issue of generating valid examples in different domains. However, there are a large number of examples that can be generated for a given concept, and it is important that examples intended for *tutoring* situations illustrate specific features that need to be communicated. There is also a strong interaction between the examples and the accompanying textual description. The number of examples to be presented, the order in which they are to be presented, and the position of the examples with respect to the text are all dependent upon the examples generated for use in the presentation. It is therefore important that the example generator be able to generate a set of suitable examples in response to a tutoring goal in a given situation. In this paper, we present one framework, based on the use of a subsumption classifier, that facilitates the generation of appropriate tutorial examples. We illustrate the working of this framework by describing the generation of examples to illustrate the syntax in the programming language LISP.

Content Areas: Tutoring Systems, Text Generation, Natural Language Front-Ends

This work was done while the first author was at the University of Southern California. The authors gratefully acknowledge support from NASA-Ames grant NCC 2-520 and DARPA contract DABT63-91-C-0025. Cécile Paris also acknowledges support from NSF grant IRI-9003087.

1 Introduction

Examples play an important role in communication and are often used in tutoring systems, e.g., [15, 14, 1, 20]. Knowledge based systems can now take advantage of advances in areas such as natural language generation and knowledge representation to generate and present coherent descriptions that integrate text and examples in an effective and natural manner, e.g., [12, 13]. There has been a lot of research on the different types of examples that should be presented in different situations for maximal effectiveness in communication – introductory vs advanced, e.g., [16], procedural vs structural, e.g., [18], etc. – but there has been relatively little work on effective computational generation of examples for use in tutorial contexts.

Early work in computational example generation was done by Michener and her colleagues, when they investigated the generation of simple examples in domains such as physics and mathematics, e.g., [21, 19]. Since then Ashley and his colleagues have studied the generation of larger and more more complex examples in domains such as Law, e.g., [1, 2]. Our framework builds upon these efforts. However, our research differs from these efforts in two important ways: the work by Michener and her colleagues investigated issues such as *how* specific examples could be generated using constraint satisfaction techniques and how pre-defined examples could be modified; while in the legal domains, the 'examples' were meant to be used as *cases*, where the selected case(s) could be used in support of arguments, but were not selected for their clarity or ease of teaching specific points. In our case, we are specifically interested in generating examples to be used in conjunction with a text generator in constructing systems for tutoring, automatic documentation, or other such related tasks. This goal, of generating tutorial examples, forces the system to construct/select examples that satisfy very specific goals: to illustrate important properties of specific features in a concept. We illustrate these requirements, and describe a framework – based on the use of a subsumption classifier – to generate suitable examples. Our discussions are illustrated with examples from our application domain of programming language syntax, mainly LISP.

2 Illustrating Critical and Variable Features with Examples

Studies have shown that user comprehension is enhanced when the examples presented contain a minimum number of irrelevant features, allowing the user to focus on the important aspects of the concept, e.g., [23]. It is therefore essential that the example generator be able to take into account the communicative goal and determine which features of the concept are to be presented. Based on the Direct Instruction Model (DIM) in educational psychology, e.g., [5, 8, 17], our system categorizes each feature of the concept into two categories, depending upon its role:

Critical Features: features that are *required* for the example to be an instance of the concept being illustrated. For instance, (as illustrated in Figure 1), the definition of a function in LISP *must* begin with the left parenthesis, followed by the keyword `defun`, followed by the function name (`F-To-C`)and a list (possibly empty) of the parameters. If either of these is missing, the expression is not a LISP function.

Variable Features: features that can change within an example without causing the example to not be an example of the concept being illustrated. For instance, the name of a function, the name and number of parameters, etc. are variable features. Their presence is critical, but their actual value is not.

The ability to categorize features as critical and variable is essential for a number of reasons. Educational research has shown that people learn critical features most efficiently when two examples are presented

A function to convert temperatures from Fahrenheit to Celsius could be written as:

```
(DEFUN F-TO-C (TEMP)
   (SETQ TEMP (- TEMP 32))
   (/ TEMP 1.8))
```

From [24], page 43.

Figure 1: An example of a function defined in LISP.

A list consists of a left parenthesis, followed by zero or more data elements, followed by a right parenthesis. Data elements can be either numbers, symbols, other lists, or a mixture of these types. For example:

```
(blue sky)        ;; a list of symbols
 blue sky)        ;; not a list
```

Figure 2: A pair of positive and negative examples can highlight critical features.

to illustrate the feature: the two examples are almost identical; they differ in *only* the critical feature. If these two examples are annotated as being positive and negative, the learner's attention can be drawn to the missing feature in the negative example. For instance, the absence of a left parenthesis in the second expression in Figure 2 causes the expression to not be a list. The pairing of the two (almost identical) examples highlights the difference, and the fact that the left parenthesis is a critical feature.

Similarly, variable features are communicated most efficiently by presenting a group of positive examples that differ widely in only the variable feature. The example generator should take these feature categorizations into account and generate either positive-negative pairs, or positive-positive groups appropriately depending upon the feature being illustrated.

Communication is further enhanced when the negative example in the positive-negative pair is not just simply labeled as a negative example, but is differentiated from the positive example. This can best be done if the negative example generated is such that it can be 'named' as an example of another, different concept. Thus, the negative example should be selected with some care. This selection will also affect the textual explanation accompanying the examples (because some additional text will have to be generated), as well as the order in which the examples will be presented (our corpus analysis has shown that the negative example with the associated explanation will be presented last). This is illustrated in the difference between the two descriptions of a list in LISP given in Figure 3.

Thus, an example generator designed for tutoring systems should be able to:

- find critical and variable features given a concept
- generate interesting negative examples (if possible) to illustrate critical features

It is important that this be done efficiently – in this paper, we describe one approach to finding critical and variable features easily using a subsumption based classifier, such as the ones available in the KL-ONE family of knowledge representation languages [4]. In addition, this also allows the system to determine whether negative examples are available or not. We have implemented this algorithm as part of a larger framework to generate system documentation (with integrated text and examples) automatically. The rest of the paper describes the algorithms used to determine the critical and variable features and illustrates them

A list consists of a left parenthesis, followed by zero or more data elements, followed by a right parenthesis. For example:

```
(blue)  ; a list of one symbol
 blue)  ; not a list
(blue   ; not a list
(4 5 6) ; a list of numbers
(3 fishes) ; a list of numbers and symbols
```

A list consists of a left parenthesis, followed by zero or more data elements, followed by a right parenthesis. For example:

```
(4 5 6) ; a list of numbers
(3 fishes) ; a list of numbers and symbols
(blue)  ; a list of one symbol
```

However, the following is not a list, *but an atom*

```
blue  ; not a list, but an atom
```

because it has no parentheses. A list differs from an atom in that ...

Figure 3: Two descriptions with a negative example.

using examples from our domain of LISP documentation.

3 Knowledge Representation

Our system is part of the documentation facility we are building for the Explainable Expert Systems (EES) Project [22], a framework for building expert systems capable of explaining their reasoning as well as their domain knowledge. In EES, a user specifies a domain model in the high level knowledge representation language LOOM [9]. LOOM is a KL-ONE type language based on description logics and supports subsumption based term classification.

To generate examples in our domain of programming language syntax, the system must first represent the BNF[1] definitions in LOOM. The conversion of a BNF form to a LOOM form is almost completely automatic.[2] An illustration of how the translation is accomplished is shown below:

```
A := B C;
```

is represented in LOOM as:

```
(defconcept A
   :is (:and B (:the grammar-sequence C)))
```

i.e., concept `A` consists of concept `B` followed by (related by the relation named `grammar-sequence` to) concept `C`; the form

[1] Backus-Naur Form.

[2] See [11] for a discussion of some of the more difficult cases in which the translation cannot be completely automated. In general, these are cases where the definition of a concept cannot be adequately described using a purely syntactic specification; in such cases, it becomes necessary to annotate the Loom definition with the predicate used in the original definition.

```
A := B | C;
```
is represented as a disjoint covering (B or C) under concept A.
```
(defconcept A
   :disjoint-covering (B C))
```
More complex definitions are represented by using a composition of the two translations shown above.

There are many advantages to using a representation such as LOOM; the main one is the availability of the classifier mechanism. As we describe in the following section, the classifier allows the generation system to do two tasks very easily: (*i*) to categorize different features in an example as being either critical or variable, and (*ii*) to determine if a negative example generated by the system is 'interesting' or not.

4 The Example Generator

4.1 Construction of Examples

Examples can either be retrieved from a pre-existing Example Knowledge Base, as in HYPO [3], or can be constructed, as in CEG [21]. Our system uses both *construction* and *retrieval* to find suitable examples. Initially, the system possesses examples of the most primitive concepts; in our case of programming syntax, the system has stored instances of grammar elements (such as `atoms`, `numbers`, `strings`, etc., in the LISP domain). Examples of such elements are therefore always retrieved. When the system needs to present an example of a more complex grammar symbol, such as a `list`, for instance, the system constructs the example based on the syntactic definition of a `list`, as well as the features being illustrated. Unlike HYPO, which used 12 pre-defined features as indices, our system uses a KL-ONE type language, LOOM *[10]* to allow us to retrieve examples with as few, or as many indices as necessary; the greater the number of indices specified in the retrieve, the fewer the number of possibilities returned by LOOM for consideration.

The example generator takes as input the list of features for a concept that needs to be illustrated by presenting an example of a particular object. The syntactic specification of the function and a typical call to it are given below:[3]

function: get-example
```
(get-example ?concept ?features ?object)
```

typical call:
```
(get-example 'data-element '(atom number) 'list)
```

In the function above, `?concept` refers to the concept being illustrated, `?features` specify the features of the concept that the example should try and illustrate, and `?object` is the object whose example should be presented. Thus, in the instantiated function call shown above, the system constructs an example *of a list*, where the concept to be illustrated is that of *a data element*, and the features that need to be highlighted are the facts that a data element can be either an *atom or a number*. The resulting output from such a function call would be

[3] In the actual implementation, the function takes other arguments as well; these are to do with the variable that needs to be instantiated with the example, etc.

```
(oranges 5)
```

The function accesses global constraints such as the text type to determine the type of elements required; in the case of the advanced text type, the element representing the number could have been a more complex, floating point number (this is done by specifying default types for the text type: lacking any further information, if a number is required for use in an example in an advanced text, the system will retrieve a floating point number, as opposed to an integer).

If the system is successful in generating an appropriate example, that example is then stored in LOOM as an example for the concept. Given the classification facility in LOOM, this is automatically indexed underneath the concept of a `list`, as well as any other grammar symbols it is applicable to. The next time the system needs to generate an example for the same features, the system can retrieve this example, rather than constructing one from scratch.

However, the function `get-example` is a relatively low level function. It takes as input both the concept and the feature being illustrated. To determine which of the features need be illustrated, it is first necessary to determine the critical and variable ones.

4.2 Determining Critical Features

As we mentioned earlier, it is essential to convey to the user that some of the concept features are *required* for any instance to be an example of the concept. These features are referred to as *critical* features. To be able to emphasize the critical nature of a feature, the system can present a pair of examples, one positive and one negative, identical in all respects, except for the critical feature being emphasized.

In our system, the representation of the domain model in LOOM allows us to determine critical features relatively easily. This is because the classification facility in LOOM allows the system to query it regarding relationships between concepts and instances. This allows the system to determine whether a particular feature is critical or not, by simply modifying the value of each feature along various dimensions and then querying LOOM to see if the modified instance still classifies as an instance of the original object. We have defined for our domain a number of ways to modify the definition of a concept.[4] The system successively attempts these operators on the given concept definition, and finds those features whose modification causes the example to fail to classify under the object being explained. The modifications attempted by the system are given in Figure 4.

The generate-and-test approach taken by the system to determine whether a particular feature is critical or not is inefficient compared to say an alternative approach based on analytically examining the LOOM definition and determining the features from there. This, however, is not possible in all cases, because as we mentioned earlier, a small number of specialized cases cannot be represented using only the LOOM semantics, but require an annotation on the concept and the use of a LISP predicate.[5] These annotations thus cannot be examined analytically to determine the critical and variable features, and it is therefore necessary to use the generate-and-test approach to classify the features as such.

[4] It is possible that these modifications will not be applicable in many domains; the alternative (to not using such domain specific modification information) is to use a representation as in CEG [21], in which every concept contained annotations on how various features could be modified.

[5] This does not, however, lessen the usefulness of this approach, because classification and querying mechanisms still function as usual on these augmented definitions.

For each feature in the set of input features, determine if the feature is a critical feature by creating an instance of a modified definition and checking whether the (modified) instance classifies under the original definition. The modified definitions are created by varying each feature in the definition as follows:

1. *Varying the Number:*

   (a) modify the definition by omitting the feature from the definition

   (b) modify the definition by adding another symbol of the same type as the current symbol in the definition

   if in either of these two cases, the modified instance fails to classify under the original definition, mark the feature as being critical along the number dimension.

2. *Varying the Type:*

   if the feature is a terminal symbol:

   (a) modify the definition by substituting the feature with another terminal symbol of the same type

   (b) modify the definition by substituting the feature with a terminal symbol of another type

   else if the feature is a non-terminal symbol:

   (a) modify the definition by substituting the feature with the superconcept of the feature

   (b) modify the definition by substituting the feature with the subconcept of the feature

   (c) modify the definition by substituting the feature with the sibling concept of the feature

   if in any of these cases, the modified instance fails to classify under the original definition, mark the type of the feature as a critical feature.

Figure 4: Determing the Critical Syntactic Features of a Concept.

There are a total of seven ways along two dimensions with which the system attempts to modify each feature of a concept definition in this domain to try and find a critical feature. Two of the seven ways in which the systems attempts modification are with respect to the *number* dimension; the remaining five are with respect to the *type* dimension. We shall illustrate the working of the algorithm by taking the example of the concept `list` in the LISP domain. The system retrieves the syntactic features of a list from the definition: the left parenthesis, the data elements, and the right parenthesis. Given these three features, the system must now determine which of these features are critical and which are variable. The system attempts to generate and test different instances created from modifying the definition of a `list`. As stated above, the system attempts to modify features along two dimensions:

Number Dimension: First, the system attempts to see if *deleting* the feature under consideration from the definition causes the system to classify this modified instance wrongly. If it does, the feature is marked as being critical. Secondly, the system checks to see whether *adding* an extra element identical to the feature causes the system to find the modified instance as belonging to another class. In both these cases, the fact that the feature is critical with regard to the number is noted by the system. Thus, if the definition is of the form

```
A -- grammar-seq -- B -- grammar-seq -- C
```

the system successively considers modified definitions of the form:

```
A -- grammar-seq -- A -- grammar-seq -- B -- grammar-seq -- C
A -- grammar-seq -- B -- grammar-seq -- B -- grammar-seq -- C
   .
   .
   .
```

the system checks to see if in any of these cases, the modified concept description still classifies as a subconcept of the original concept.

For the case of a `list`, instances of a `list` are created from modified definitions and tested to see whether they classify under the original definition of a `list`. Modifications along the number dimension, such as reducing the number of parentheses by one, or adding an extra parenthesis, cause the instances to not classify under the original definition. Thus, *both the left and the right parentheses are marked as critical.* On the other hand, modifications to the number of data elements in the list, by either deleting one, or adding one, do not result in the instance failing to classify as a list. At this point therefore, the number of data elements are not classified as critical features.

Type Dimension: There are a number of different ways in which the system attempts to modify a feature by varying the type dimension:

Terminals: If the feature being considered happens to be a terminal symbol, the system modifies the definition of the concept in two ways: (i) by replacing the symbol with another terminal symbol of the same type. For instance, if the terminal symbol happened to be a number, say 2, the system would try to replace 2 with another number, for instance, 7. (ii) by replacing the terminal symbol with another terminal symbol of another type. For instance, in the previous case, the system could attempt to replace the number 2 with a character, such as `'a'`. In the case of the `list`, the system can attempt to replace the `left-parenthesis` with another terminal symbol, such as the `right-parenthesis`, and in the second case, by a keyword, such as `'defun'`. If in either of these cases, an instance of the modified definition did not classify as an instance of the original definition, the system would mark the fact that the type of the feature was a critical feature.

Non-Terminals: If the feature being considered is a non-terminal symbol, the system attempts to modify the definition by changing the symbol in three different ways: (i) by replacing it with a superconcept, (ii) by replacing it with a sub-concept, and (iii) by replacing it with a sibling concept. In the case of the `list`, case (i) is not applicable, because `data-element` is the most general type in the representation of a `list`, since it is the disjunction of the `symbol`, `number`, and `list` types; case (ii) could result in the system replacing `data-element` with another type such as `number`, and case (iii) is again not applicable in the case of `data-element`. Since a list of numbers is still a valid list, the type aspect of `data-elements` is not marked as being a critical feature for a `list`. The algorithm allows the system to determine the critical features of a concept. Once these features have been determined, the system caches these values so that it does not have to repeat this reasoning the next time it has to determine critical features for the same object and is given the same set of input features.

As in the case of `get-example`, the function to find the critical features of an object has been designed so that it can be invoked within the tutoring system and the results of the function call used in other reasoning as well. The function is given a list of features and an object, and returns those features from the set that are critical. A typical call is shown below:

function: select-critical-features

```
(select-critical-features ?features ?object)
```

typical call:

```
(select-critical-features
   '(left-parenthesis (kleene-closure data-elements)
                      right-parenthesis)
   'list)
```

In this case, the function call returns:

```
(left-parenthesis right-parenthesis)
```

The function *selects* critical features from a list of features passed to it, rather than *finding* the critical features, because different cases may require the presentation of different sets of features. For instance, the generation of descriptions for introductory and advanced users requires the presentation of quite different amounts and types of information in many domains. Thus, in our system, the text generator first selects the appropriate features for the given user type from the LOOM representation, and then, determines the critical features from this set of features to be presented.

4.3 Determining Variable Features

As in the case of critical features, the system must know which features are variable in nature. A knowledge of the variable features then allows the system to illustrate the variability by presenting multiple positive examples that vary in the variable features. Since variable features are not critical features, if the critical features for a concept are known, the system can attempt to prune the set of features to be considered by removing the critical features.[6] The remaining features are then processed exactly in the same manner in which the critical features are determined; the only difference is that the systems tests for successful classification (rather than a failure to classify) after each modification. Each feature is varied along both the type, and the number dimensions as in the previous case regarding the critical features:

- **Number Dimension:**
  - vary the definition by omitting the current feature from the definition
  - vary the definition by adding another feature of the same type as the current feature
- **Type Dimension:**
  - if feature is a terminal: attempt replacements with (i) other terminals of the same type, and (ii) terminals of another type
  - if feature is a non-terminal: attempt replacements with subtype, supertype and sibling types

[6]The reasoning mechanism which determines the critical features also uses this null intersection criteria to prune the set of features it has to consider in finding critical features.

If instances created from the modified definitions still classify under the original definition of the concept, the feature is marked appropriately as a variable feature. As we mentioned previously, LOOM allows us to determine the class of the description very simply with its classification mechanism. As in the case of critical features, the variable features of the object are cached upon computation so that future calls to the function can be answered using simple retrieves.

Features of a concept can be critical and variable at the same time–along different dimensions. Consider the case of the operator PLUS in LISP for instance. While the number of arguments that follow the operator are not critical, the type of the arguments is–they should be numbers. Similarly, in the case of the operator CONS in LISP, the number of arguments is critical, while their type is not. It is therefore important to identify not just whether a feature is critical or variable,[7] but also in what respect.

4.4 Finding Interesting Negative Examples

An important aspect in generating tutorial descriptions is the presentation of negative examples. Negative examples need to be presented to highlight the critical aspects of the concept being described. However, since there can be different negative examples that can be used in any given situation, it is beneficial to use examples that are 'interesting' in some sense, rather than any random example. Consider for instance, the case of a `list` in Figure 3. Both the descriptions emphasize the critical nature of the parentheses. However, the second description is more pedagogical, because it conveys not only the fact that the negative example is not a list, but *also* that it is an atom. It is therefore important to find such 'interesting' negative examples, if they are available. Note also that this allows the system to opportunistically include more material if so desired on the differences between the two concepts.

In our system, finding interesting negative examples is made easy using the classification mechanism in LOOM. Each time the system finds a critical feature, it tests to see whether the modification causing the example to become negative also causes the example to classify under *another* description in the knowledge base. If it does, the system marks this critical feature, as well as the classification of the negative example, and uses this in preference to some other example.

This method of finding interesting negative examples is very dependent on the availability of a classification mechanism.[8] While the previously mentioned use of LOOM (in determining critical and variable features) could possibly be implemented even without the use of a classifier, finding interesting negative examples would be much harder to implement without this capability.

5 A Trace of the System

In this section, we shall illustrate how the algorithms described in the previous section generate a set of examples to be used in illustrating the concept of a `list` in LISP. The use of examples in teaching is not a trivial task; each example must be clear and easy to understand. In addition, the position of the example in

[7] All features are either critical or variable, depending upon their role in the concept definition. However, some critical features such as parentheses in LISP are so ubiquitous that they can be a distraction when discussing complex constructs. To handle this aspect, the system also possesses the concept of *fixed features*, which are critical features and therefore appear in all examples, but are not explicitly used by the system to generate negative examples, or commented upon. For further details, please see [11].

[8] Classification, or structural subsumption, is theoretically undecidable [7]. However, in practice, exponential algorithms exist that will determine for certain restricted languages whether one description logically entails another.

a sequence may indicate relationships between the examples that are important for the learner to grasp: for instance, the critical and variable features are indicated by pairing positive-negative and positive-positive examples appropriately. We have already seen how a good negative example found using a subsumption classifier can be used effectively to illustrate a critical feature. In this section we shall elaborate on the example generator by showing how variable features can be highlighted through the presentation of carefully planned and presented groups of positive examples.

In the case of a `list` in LISP, the example generator determines that the variable features are the *type* and the *number* of `data-elements` in a `list`.[9] The example generator must generate an appropriate sequence of examples to illustrate this fact. Since data elements can vary in two dimensions, it posts two internal goals, one for each varying dimension. The goal to illustrate the variable number of data elements causes the posting of two goals, one to generate an example with a single element, and one to generate an example with multiple (four) elements.

```
(GENERATE-EXAMPLE (VAR-FTR DATA-ELEMENT) 1 LIST)
(GENERATE-EXAMPLE (VAR-FTR DATA-ELEMENT) 4 LIST)
```

Note that the system picks the numbers 1 and 4 for the following reasons: the system needs to pick an example at the lower end of the range of possible numbers, and selects zero, but a list with no elements is defined as the symbol `NIL` as well. Since the symbol `NIL` classifies as an anomalous example, the system chooses 'one' as the number of elements to present. At the other end of the range, 'four' is selected because of a heuristic in the system as a typical higher limit. Both of these goals causes other goals to be posted to actually construct the example. The example generation algorithm ensures that (i) the examples selected for related sub-goals (such as the two above) differ in *only* the dimension being highlighted; (ii) the remaining dimensions are kept as simple as possible: thus the examples generated contain only atoms. (Both numbers and atoms are considered to be equally complex in this implementation, and numbers could also have been chosen to construct the three simpler lists; however, the implementation in LOOM returns the first of the retrieved list, and this happens in this case to be atoms.) The resulting output of these two goals is the presentation of two lists of atoms, one with a single element, and another with four elements.

Similarly, the goal to illustrate the type variability of elements in a list causes the generation of multiple goals: a goal to illustrate the fact that data elements can be atoms, numbers, number+atoms, lists, numbers+lists, etc. The fact that there exists a kleene-closure of the data-elements causes the system to generate a power-set of all the sub-types. These are then sorted in order of increasing complexity.[10] The first four goals to present examples are selected. This is based on Clark's maxim of four examples [6] as being sufficient to convey a variable feature in most cases. These four goals are:

```
(GENERATE-EXAMPLE (VAR-FTR ATOM) LIST)
(GENERATE-EXAMPLE (VAR-FTR NUMBER) LIST)
(GENERATE-EXAMPLE (VAR-FTR (ATOM NUMBER)) LIST)
(GENERATE-EXAMPLE (VAR-FTR LIST)) LIST)
```

The presentation sequence ensures that only one feature is modified between each adjacent example, thus

[9] As we have noted earlier, in different contexts, the type and the number attributes of a concept can be independently either critical or variable; in this case, both of them are variable.

[10] The complexity algorithm is application specific and in this case assigns a complexity value based on the number of grammar productions required to generate an example. See [11] for details.

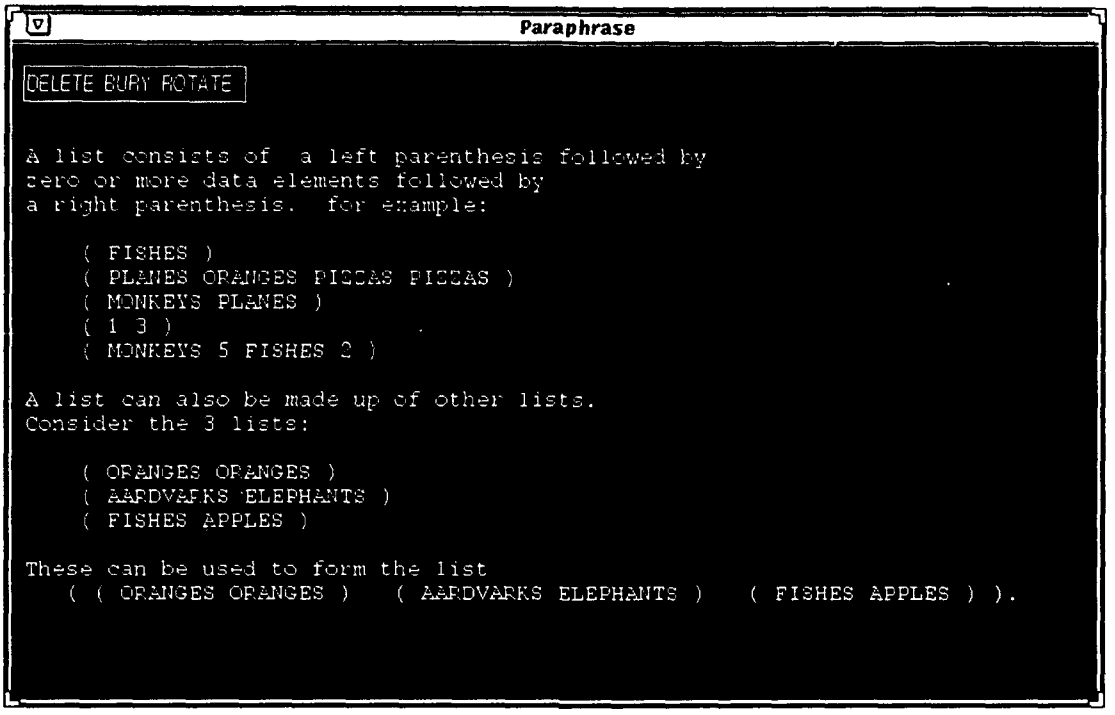


Figure 5: A snapshot of the output generated for a `list` in LISP.

drawing the learner's attention to that feature– in this case, the type of the `data-elements`.

The final goal – of presenting a list of lists – poses a difficult problem to the generator. This is because in a tutorial case, the system cannot simply present examples of either recursive or anomalous cases without explicitly marking them as such: this is done through the presentation of information explaining such concepts to the user. The system therefore posts two goals, one to provide background information (which presents some simple lists), and the other to build a list from these three lists. Since these lists need to be as simple as possible, the previously presented lists which varied in their number of elements as well as their type, are not selected for re-use. It therefore generates three lists that have only atoms as their `data-elements`. These lists are then used to construct the recursive case. A screen snapshot of the actual examples generated by the system (together with the accompanying explanation generated by the text planner) is shown in Figure 5.

6 Conclusions

In this paper, we have described one framework for generating examples suitable for use in tutorial contexts. In order to do this, the system must first determine the critical and variable features of a concept and use this knowledge in generating appropriate sequences of examples. We have presented algorithms that can automatically categorize features based on a representation of the syntactic definition in a KL-ONE type language. We have implemented this system using LOOM; however, the methods are not specific

to LOOM and can be implemented using any subsumption based classifier. The system has been used to generate examples of relatively complex concepts in both LISP and another language designed here, known as INTEND.

References

[1] Vincent Aleven and Kevin D. Ashley. Automated generation of examples for a tutorial in case-based argumentation. In *Proceedings of the Second International Conference on Intelligent Tutoring Systems (ITS-92)*, Montreal, Canada., 1992.

[2] Kevin Ashley and Vincent Aleven. Generating dialectical examples automatically. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 654–660, San Jose, CA., 1992. American Association for Artificial Intelligence.

[3] Kevin D. Ashley. Reasoning with cases and hypotheticals in HYPO. *International Journal of Machine Studies*, 34(6):753–796, June 1991.

[4] Ronald Brachman. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9:171–216, 1985.

[5] Jerome S. Bruner. *Toward a Theory of Instruction*. Oxford University Press, London, U.K., 1966.

[6] D. C. Clark. Teaching Concepts in the Classroom: A Set of Prescriptions derived from Experimental Research. *Journal of Educational Psychology Monograph*, 62:253–278, 1971.

[7] Jon Doyle and Ramesh S. Patil. Two theses of Knowledge Representation: Language restrictions, taxonomic classification, and the utility of representation devices. *Artificial Intelligence*, 48:261–297, 1991.

[8] Siegfried Engelmann and Douglas Carnine. *Theory of Instruction: Principles and Applications*. Irvington Publishers, Inc., New York, 1982.

[9] Robert MacGregor. A Deductive Pattern Matcher. In *Proceedings of the 1988 Conference on Artificial Intelligence*, St Paul, Mn, August 1988. American Association of Artificial Intelligence.

[10] Robert MacGregor. The Evolving Technology of Classification-Based Knowledge Representation Systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, San Mateo, California, 1991.

[11] Vibhu O. Mittal. *Generating Descriptions with Integrated Text and Examples*. PhD thesis, University of Southern California, Los Angeles, CA, 1993.

[12] Vibhu O. Mittal and Cécile L. Paris. Analogical Explanation in the EES Framework. In Johanna D. Moore and Michael R. Wick, editors, *Proceedings of the AAAI'90 Workshop on Explanation*, pages 162–172, Boston, MA, August 1990. American Association for Artificial Intelligence.

[13] Vibhu O. Mittal and Cécile L. Paris. Automatic Documentation Generation: The Interaction between Text and Examples. In *Proceedings of Thirteenth International Joint Conference on Artificial Intelligence*, pages 1158–1163, Chambery, France, 1993. IJCAI.

[14] Vibhu O. Mittal and Cécile L. Paris. Building Intelligent Help Facilities: Generating Natural language Descriptions with Examples. In Gavriel Salvendy and Michael J. Smith, editors, *Human-Computer Interaction: Software and Hardware Interfaces*, pages 379–384, Orlando, FL, August 1993. Elsevier.

[15] Vibhu O. Mittal and Cécile L. Paris. Categorizing Example Types in Instructional Texts: The Need to Consider Context. In Paul Brna, Stellan Ohlsson, and Helen Pain, editors, *Proceedings of AI-ED 93: World Conference on Artificial Intelligence in Education*, pages 137–144, Edinburgh, Scotland, August 1993. AACE.

[16] Vibhu O. Mittal and Cécile L. Paris. Generating Natural Language Descriptions with Examples: Differences between introductory and advanced texts. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI '93)*, pages 271–276, Washington, DC, 1993. (Also available as USC/ISI Research Report #ISI/RR-93-331).

[17] Joseph Moore. Direct Instruction: A Model of Instructional Design. *Educational Psychology*, 6(3):201–229, 1986.

[18] Peter L. Pirolli and John R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 39:240–272, 1985.

[19] Edwina L. Rissland. Example Generation. In *Proceedings of the Third National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 280–288. CIPS, Toronto, Ontario, May 1980.

[20] Edwina L. Rissland and Elliot M. Soloway. Overview of an Example Generation System. In *Proceedings of the Na' (l Conference on Artificial Intelligence*, pages 256–258. AAAI, 1980.

[21] Daniel D. Suthers and Edwina L. Rissland. Constraint Manipulation for Example Generation. COINS Technical Report 88-71, Computer and Information Science, University of Massachusetts, Amherst, MA., 1988.

[22] William R. Swartout, Cecile L. Paris, and Johanna D. Moore. Design for Explainable Expert Systems. *IEEE Expert*, 6(3):58–64, 1992.

[23] Mark Ward and John Sweller. Structuring Effective Worked Examples. *Cognition and Instruction*, 7(1):1–39, 1990.

[24] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, Reading, MA, 1984.